Scheduling with Global Information in Distributed Systems

Fabrizio Petrini and Wu-chun Feng

{fabrizio, feng}@lanl.gov

Computing, Information, & Communications Division

Los Alamos National Laboratory

Los Alamos, NM 87545, USA

August 22, 1999

Abstract

One of the major problems faced by the developers of parallel programs is the lack of a clear separation between the programming model and the operating system. In this paper, we present a new methodology to multitask parallel jobs in a message-passing environment and to develop parallel programs that can pave the way to the efficient implementation of a distributed operating system. This methodology is based on three innovative techniques: communication buffering, strobing, and non-blocking, one-sided communication. By leveraging these techniques, we can perform effective optimizations based on the gloabl status of the parallel machine rather than on the limited knowledge available locally to each processor.

The advantages of the proposed methodology include higher resource utilization, reduced communication overhead, efficient implementation of flow-control strategies and fault-tolerant protocols, accurate performance modeling, and a simplified yet still expressive parallel programming model.

Some preliminary experimental results show that this methodology is very effective in increasing the overall performance in the presence of load imbalance and communication intensive workloads.

Keywords: Paralllel Job Scheduling, Distributed Operating Systems, Communication Protocols.

1 Introduction

The scheduling of parallel jobs has long been an active area of research [10, 11]. It is a challenging problem because the performance and applicability of parallel scheduling algorithms is highly dependent upon factors at different levels: the workload, the parallel programming language, the operating system (OS), and the machine architecture.

Time-sharing scheduling algorithms are particularly attractive because they can provide good response time without migration or predictions on the execution time of the parallel jobs. However, time-sharing has the drawback that *communicating processes must be scheduled simultaneously to achieve good performance*. With respect to performance, this is a critical problem because the software communication overhead and the scheduling overhead to wake up a sleeping process dominate the communication time on most parallel machines [20].

Over the years, researchers have developed parallel scheduling algorithms that can be loosely organized into three main classes, according to the degree of coordination between processors: *explicit coscheduling*, *local scheduling* and *implicit or dynamic* *coscheduling*.

On the one end of the spectrum, explicit coscheduling [9] ensures that the scheduling of communicating jobs is coordinated by constructing a static global list of the order in which jobs should be scheduled. A simultaneous context-switch is then required across all processors. Unfortunately, these straightforward implementations are neither scalable nor reliable. Furthermore, explicit coscheduling requires that the schedule of communicating processes be precomputed, which complicates the coscheduling of client-server applications and requires pessimistic assumptions about which processes communicate with one another. Finally, explicit coscheduling of parallel jobs interacts poorly with interactive jobs and jobs performing I/O [21].

At the other end of the spectrum is local scheduling, where each processor independently schedules its processes. This is an attractive time-sharing option due to its ease of construction. However, the performance of fine-grained communication jobs can be orders of magnitude worse than with explicit coscheduling because the scheduling is not coordinated across processors [14].

An intermediate approach initially developed at UC Berkeley and MIT in recent years is implicit or dynamic coscheduling [1, 24, 8, 33]. With implicit coscheduling, each local scheduler makes independent decisions that dynamically coordinate the scheduling actions of cooperating processes across processors. These actions are based on local events that occur naturally within communicating applications. For example, on message arrival, a processor speculatively assumes that the sender is active and will probably send more messages in the near future. The implicit information available for implicit coscheduling consists of two inherent events: *response time* and *message arrival* [1]. An in-depth performance analysis of coscheduling strategies is reported in [23].

Response time is the time for the response to a message request to return to the sending process. Assuming the destination process must be scheduled for a response to be returned, a fast response indicates to the sending node that the corresponding destination process is probably currently scheduled. Therefore, the desired action for implicit coscheduling is to keep the sender scheduled. Conversely, if the response is not received in a timely fashion, the sending node can infer that the destination is probably *not* scheduled. Thus, it is not beneficial to keep the sender scheduled.

The mechanism that achieves these desired actions is *two-phase spin blocking*. With two-phase spin-blocking, a process spins for some threshold amount of time, and if the response arrives before the time expires, it continues executing. If the response is not received within the threshold, the process voluntarily relinquishes the processor so a competing process can be scheduled.

The other inherent event used in implicit coscheduling is message arrival, the receipt of a message from a remote node. When a message arrives, the implication is that the corresponding remote process was recently scheduled. Therefore, it may be beneficial to schedule, or keep scheduled, the receiving process and to increase its spin time.

The main drawbacks of dynamic and implicit coscheduling include (1) the limited programming model supported, (2) the limitation of a localized flow-control strategy, (3) the non-trivial implementation of fault tolerance, and (4) the lack of a reliable performance model of the execution time of parallel jobs, due to the dynamic interleaving of several jobs. We elaborate on a few of these drawbacks below.

In the presence of fine-grained communication, implicit coscheduling increases the spinning threshold. As a consequence, many processes speculatively spin waiting for message arrival, potentially wasting CPU time with jobs having irregular communication patterns. In addition, implicit coscheduling incurs communication overhead on a per-message basis.

The programming model used in the implementation of implicit coscheduling does not support a full-fledged communication library as MPI and considers only three basic communication operations: *reads* and *writes*, request-response messages between pairs of processes requiring the requesting process to wait for the response, and *barriers* to synchronize all processes.

The limitation of using a localized flow-control strategy emerges when processes perform continuous reads or writes in an irregular communication pattern. In this case, they can flood the output buffers with write operations [1].

Some of these limitations are successfully addressed in [23], with a technique called *Periodic Boost*. Rather than sending an interrupt for each incoming message, the kernel periodically examines the status of the network interface, thus reducing the overhead with high communication workloads. Also, the experiments reported in [23] consider a complete implementation of MPI. Our methodology is based on a similar buffering technique, which is integrated with the notion of a global time-slice, and the strobing algorithm.

In this paper we present a new methodology that tries to conjugate the positive aspects of explicit and implicit coscheduling using three innovative techniques: communication buffering to amortize communication overhead (a technique very close to Periodic Boost); strobing to globally exchange information at regular intervals; and non-blocking, one-sided communication to decouple communication and synchronization. By leveraging these techniques, we can perform effective optimizations based on the status of the parallel machine rather than on the limited knowledge available locally to each processor.

The benefits of the proposed methodology include higher resource utilization, a dramatic simplification of the run time support, reduced communication overhead, efficient global implementation of flow-control strategies and fault-tolerant protocols, accurate performance modeling, and a simplified yet still expressive parallel programming model (*a la* CISC$\rightarrow$RISC instruction-set simplification).

The rest of the paper is organized as follows. Section 2 characterizes important properties which are shared by many parallel applications, e.g., resource utilization and communication access patterns, and which inspired our proposed methodology. The methodology itself is described in Section 3 and some preliminary results are presented in Section 4. Finally, we present our conclusions in Section 5.

2 Resource Utilization of Parallel Programs

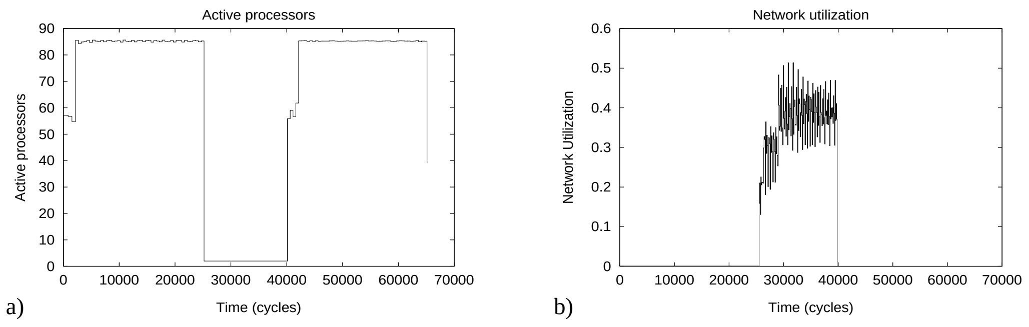

Figure 1: Resource Utilization in a Transpose FFT Algorithm.

In Figure 1, we show the *global* processor and network utilization (i.e., the number of active processors and the fraction of active links) during the execution of a transpose FFT algorithm on a parallel machine with 256 processors. These processors are connected with an indirect interconnection network using state-of-the-art routers [30]. Based on these figures, there is obviously an *uneven and inefficient use of system resources.* During the two computational phases of the transpose, the network is idle. Conversely, when the network is actively transmitting messages, the processors are not doing any useful work. These characteristics are shared by many SPMD programs, including Accelerated Strategic Computing Initiative (ASCI) application codes such as Sweep3D [16]. Hence, there is tremendous potential for increasing resource utilization in a parallel machine.

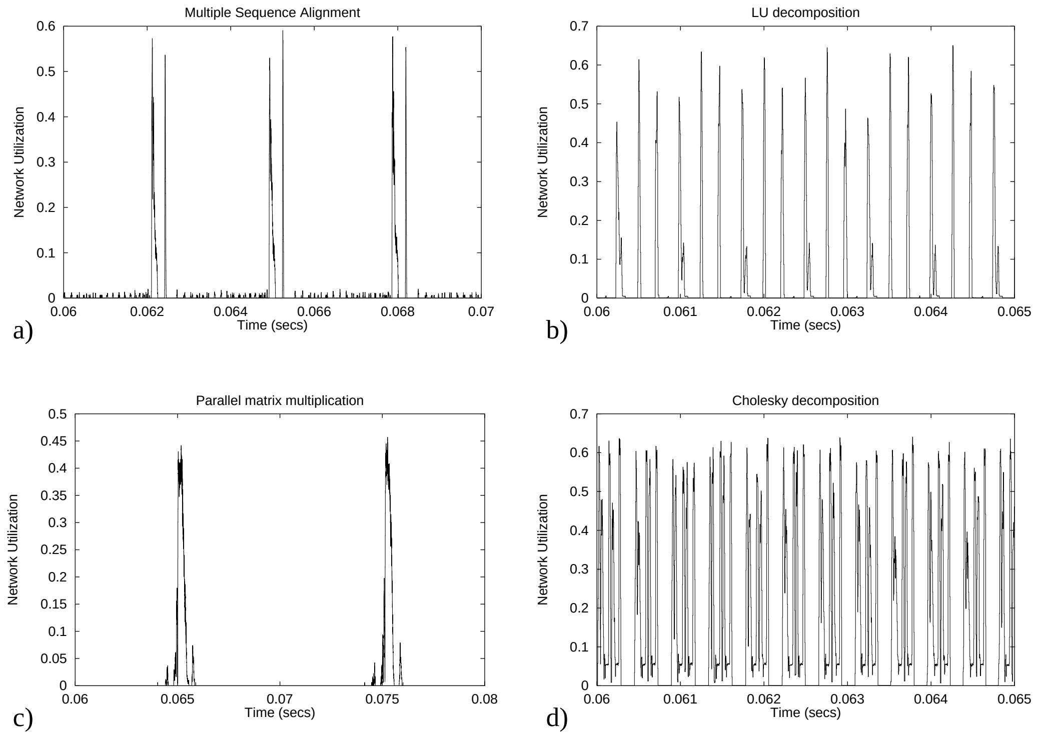

Figure 2: Network Utilization in Scientific Parallel Programs.

Another important characteristic shared by many parallel programs is their access pattern to the network. The vast majority of parallel applications display *bursty communication patterns* with alternating spikes of impulsive communication with periods of inactivity.

In Figure 2, we show the network utilization by running four distinct applications over a parallel machine with 256 processors [26]. In all four cases, we can identify *communication holes*, i.e., periods of network inactivity, in the network. Therefore, there exists a significant amount of communication bandwidth which can be made available for other purposes.

3 Multitasking Parallel Jobs

In order to improve the resource utilization of parallel programs, we propose to multitask parallel jobs. That is, *instead of overlapping computation with communication and* I/O *within a single parallel program, all the communication and* I/O *which arises from a set of parallel programs can be overlapped with the computations in those programs.*

We propose three techniques to implement the multitasking of parallel jobs.

1. The communication generated by each processor is buffered and performed at the end of regular intervals (or time-slices) in order to amortize the communication and scheduling overhead. By delaying the communication, we allow for the global scheduling of the communication pattern.

2. A strobing mechanism performs a total exchange of information at the end of each time-slice so that multiprocessor machines may move away from isolated scheduling algorithms [1, 24, 8, 33] (where processors make decisions based solely on their local status and a limited view of the remote status) to more outward-looking or global scheduling algorithms.

3. Finally, we propose to use non-blocking and one-sided communication primitives to decouple communication and synchronization in order to schedule the communication pattern with additional degrees of freedom.

This approach represents a significant improvement over existing work reported in the literature. It allows for the implementation of a global scheduling policy, as done in explicit coscheduling, while maintaining the overlapping of computation and communication provided by implicit coscheduling.

3.1 Communication Buffering

Rather than incurring communication and scheduling overhead on a per-message basis, we propose to accumulate the communication messages generated by each processor and amortize the overhead over a set of messages. Specifically, the cost of the system calls necessary to access the kernel data structures for communication is amortized over a set of system calls rather than being incurred on each individual system call. This implies that our methodology can be tolerant to the potentially high latencies that can be introduced in a kernel call or in the initialization of the NIC that can reside on a slow I/O bus. In addition to amortizing communication and scheduling overhead, we can also implement zero-copy (or low-copy, if we desire fault-tolerant communication) communication. As a result, our approach to communication buffering can achieve performance comparable to user-level network interfaces (i.e., OS-bypass protocols) [5, 22, 2, 17, 18] without using specialized HW.

3.2 Strobing

The uneven resource utilization and the periodic, bursty communication patterns generated by many parallel applications can be exploited to perform a total exchange of information and synchronizing the processors at regular intervals with little additional cost. This provides the parallel machine with the capability of filling in communication holes generated by parallel applications.

In order to provide the above capability, we propose a strobing mechanism to support the scheduling of a set of parallel jobs which share a parallel machine. Let us assume that each parallel job runs on the entire set of p processors, i.e., jobs are time-sharing the whole machine. The strobing mechanism performs an optimized total-exchange of control information and also triggers the downloading of any buffered packets into the network. The strobe can be implemented by designating one of the processors as the *master*, the one who generates the "heartbeat" of the strobe. The generation of heartbeats will be achieved by using a timeout mechanism which can be associated with the network interface card (NIC). This ensures that strobing incurs very little CPU overhead as most NICs can count down and send packets asynchronously. This is true for a wide range of NICs, ranging from simple 100-Mbit Ethernet cards to more sophisticated devices such as the Myrinet cards [3].

On reception of the heartbeat, each processor (excluding the master), is interrupted and downloads a broadcast heartbeat into network. After downloading the heartbeat, the processor continues running the currently active job. (This ensures computation is overlapped with communication.) When p heartbeats arrive at a processor, the processor will enter a strobing phase where its kernel will download any buffered packets. Each heartbeat contains information on which processes have packets ready for download and which processes are asleep waiting to upload a packet from a particular processor.

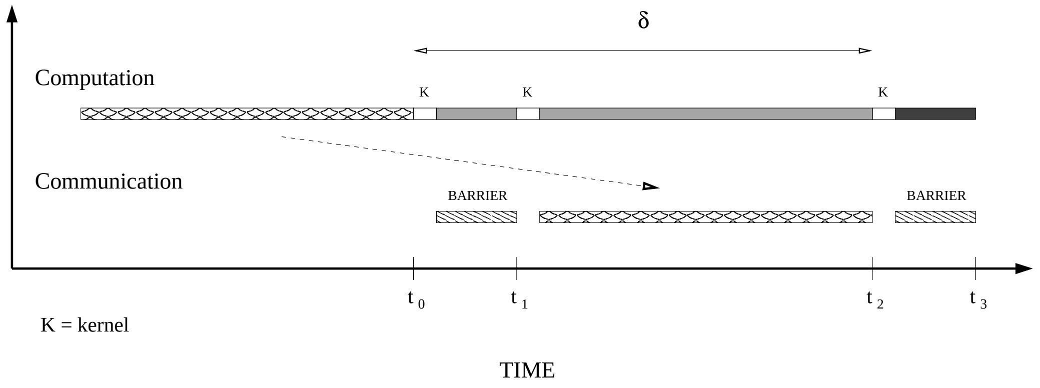

Figure 3: Scheduling Computation and Communication. The communication accumulated before t_0 is downloaded into the network between t_1 and t_2 (after the completion of the barrier synchronization).

Figure 3 outlines how computation and communication can be scheduled over a generic processor. At the beginning of the heartbeat, t_0, the control is given to the kernel, which downloads the control packets for the total exchange. During the execution of the barrier synchronization, the user process regains control of the processor, and at the end of it, the kernel schedules the pending communication accumulated before t_0 to be delivered in the next time-slice. At t_1, the processor will know the number of incoming packets that it is going to receive in the next communication time-slice as well as the sources of the packets and will start the downloading of outgoing packets.

This strategy can be easily extended to deal with space-sharing where different regions run different sets of programs [9, 34, 19]. In this case too, all these regions are synchronized by the same heartbeat.

The total-exchange can be properly optimized by exploiting the low-level features of the interconnection network. For example, if control packets are given higher priority than the standard background traffic at the sending and receiving endpoints, they can be delivered with predictable latency.

In Figure 4 we analyze the network latency[1] distribution of the control packets during the execution of a direct total-exchange algorithm[2] [27]. In this simulation, we consider a network with 256 processing nodes equipped with wormhole routers similar to those of the SGI Origin 2000 [7, 6, 12], and we assume the existence of a background random traffic that occupies 80% of the network capacity. If the control packets are prioritized at the network endpoints they can be delivered with a bounded latency, below 30 μsec.

We also analyzed the execution time of the direct total-exchange algorithm in a family of indirect networks with up to 1024 processing nodes. In this experiment,

[1]The network latency is the time spent in the network without including source and destination queueing delays.

[2]In a direct total-exchange algorithm each packet is sent directly from source to destination, without intermediate buffering.

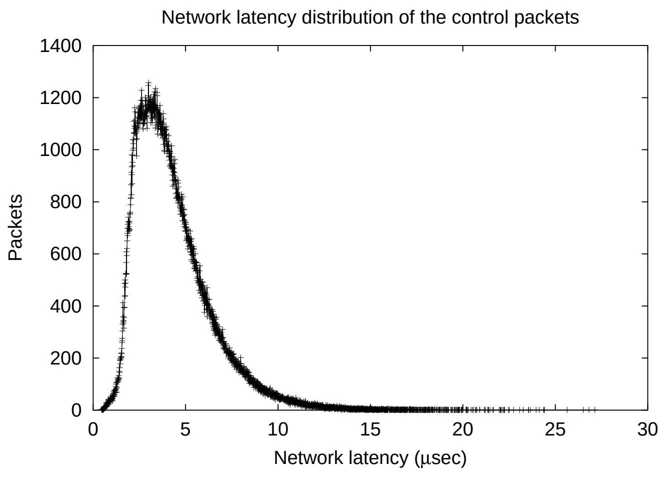

Figure 4: Network latency distribution of the control packets in a network with 256 processing nodes.

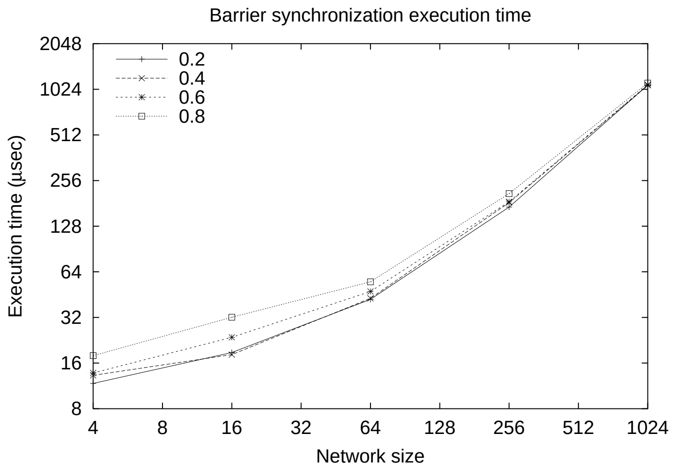

Figure 5: Execution time of the total exchange algorithm in a family of interconnection networks with up to 1024 processing nodes.

whose results are shown in Figure 5, we assume the existence of background traffic that varies from 20% to 80% of the network capacity. We can see that the execution time is largely insensitive to the intensity of the background traffic. With 64 processing nodes (the configuration of a single SGI Origin 2000 cluster) the execution time is only 50 μsec and this increases to 150 μsec with 256 nodes. Due to the quadratic increase of the number of messages sent during the total-exchange, the execution time reaches 1 msec with 1024 nodes, limiting the scalability of the approach. This problem can be addressed in a clustered architecture, like ASCI Blue Mountain, by using a multi-phase, indirect algorithm, that in the first phase executes the total-exchange inside each single cluster, then performs a total-exchange between clusters, to conclude with a final phase internal to the clusters, giving a barrier synchronization time of less than 300 μsec.

The global knowledge of the communication pattern provided by the total exchange allows for the implementation of efficient flow-control strategies. For example, it is possible to avoid congestion inside the network by carefully scheduling the communication pattern and limiting the negative effects of hot spots by damping the maximum amount of information addressed to each processor during a time-slice. The same information can be used at the kernel level to provide fault-tolerant communication. For example, the knowledge of the number of incoming packets greatly simplifies the implementation of receiver-initiated recovery protocols.

3.3 Blocking vs. Non-Blocking

One of the most limiting constraints in the implementation of time-sharing algorithms is the need to schedule simultaneously communicating processes. This problem is exacerbated with blocking communication, which imposes an explicit handshake between sender and receiver.

We argue that this problem can eliminated, or at least alleviated, by slightly modifying the communication structure of parallel jobs and replacing blocking communication with non-blocking primitives and/or one-sided communication.

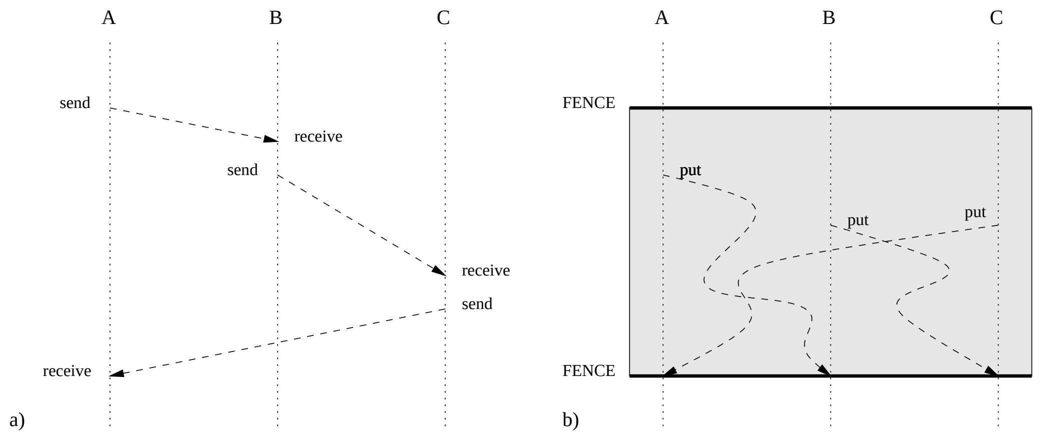

Figure 6: (a) Message Passing. (b) One-Sided Communication.

Let us consider the following example. The dynamics of a message-passing program can be represented as a two-dimensional graph with processes on the horizontal axis and time on the vertical one, as shown in Figure 6. Arrows between processes represent communication between a sender and a receiver. In Figure 6(a), three processes exchange messages. For the sake of convenience, let us assume that there is no dependency between the messages (i.e., they can be sent in any order). Using a traditional, blocking, message-passing programming style, we must define a communication schedule even if one is *not* required, e.g., A sends to B, B receives from A and sends to C, C receives from B and sends to A.

With one-sided communication (or non-blocking communication primitives, in general), the actual message transmission and the synchronization are decoupled, leaving many degrees of freedom to re-arrange message transmission. In Figure 6(b), the same communication pattern is delimited by two *barriers* which include the communication executed with *put* primitives. The communication can be executed in any order, provided that the information is delivered at the end of the synchronization calls. Also, communicating processes do *not* need to be simultaneously scheduled to perform the communication.

3.4 Bulk-Synchronous Parallel Programs

Using our proposed strobing and buffering mechanisms, any generic parallel program can be transformed into a Bulk-Synchronous Parallel (BSP) one [32]. Although the buffering and strobing mechanisms alone improve parallel program performance, transforming by themselves a parallel program into a BSP one not only can improve performance further but also allows for accurate prediction of the execution times.

A BSP computation consists of a sequence of parallel *supersteps*. During a superstep, each processor can perform a number of computation steps on values held locally at the beginning of the superstep and can issue various remote read and write requests that are buffered and delivered at the end of the superstep. This implies that communication is clearly separated from synchronization, i.e. it can be performed in any order, provided that the information is delivered at the beginning of the following superstep. However, while the supersteps in the original BSP model can be variable in length, our programming model generates computation and communication slots which are fixed in length and are determined by the time-slice.

One important benefit of the BSP model is the ability to accurately predict the execution time requirements of parallel algorithms and programs. This is achieved by constructing analytical formulae that are parameterized by a few constants which capture the computation, communication, and synchronization performance of a p-processor system. These results are based on the experimental evidence that the generic collective communication pattern generated by a superstep called h-relation[3]

[3] h denotes the maximum amount of information sent or received by any process during the

can be routed with predictable time [13, 28]. This implies that the maximum amount of information sent or received by each processor during a communication time-slice can be statically determined and enforced at run time by a global communication scheduling algorithm. For example, if the duration of the time-slice is δ and the permeability of the network (i.e., the inverse of the aggregate network bandwidth) is g, the upper bound h_{max} of information, expressed in bytes, that can be sent or received by a single processor is

$$h_{max} = \frac{T}{g}.$$

Furthermore, by globally scheduling a communication pattern, as described in Section 3.2, we can derive an accurate estimate of the communication time with simple analytical models already developed for the BSP model [28, 4, 27].

Another important benefit of the BSP model is higher resource utilization over the parallel machine, irrespective of the computational and communication patterns. For example, a sparse communication pattern (where a single processor receives h_{max} bytes) or a more dense communication pattern (where more processors share the same upper bound) can be routed in the same communication time-slice. This means that it is possible to use spare communication bandwidth to deliver packets generated by other parallel jobs, without detrimental effects. More generally, as with any multiprogrammed system, multitasking a collection of bad (parallel) programs, i.e., unbalanced computation or communication, may produce the same behavior as a single well-behaved (parallel) program. Multitasking can provide opportunities for filling in "spare communication cycles" by merging sparse communication patterns together to produce a denser communication pattern.

Lastly, the BSP model is also beneficial for fault tolerance[4] Fault tolerance can

superstep.

[4]This is of vital importance to the large ASCI supercomputers where the MTBF can be on the order of hours.

be enhanced by exploiting the synchronization points at the end of a time-slice: we can take a snapshot of the whole machine and checkpoint its status.

4 Experimental Results

Our preliminary experimental results include a working implementation of a representative subset of MPI-2 on a detailed (register-level) simulation model [29]. The simulation environment includes a standard version of MPI-2 and a multitasking one, that implements the main features of our proposed methodology.

Because the design space of our problem is too large to explore exhaustively, we fix the workload and system characteristics.

4.1 Characteristics of the Synthetic Workloads

The workloads used consist of a collection of single-program multiple-data (SPMD) parallel jobs, as reported in [8], that alternate phases of purely local computation with phases of interprocess communication. A parallel job generated by one of such programs consists of a group of P processes and each process is mapped on a processor throughout the execution. Processes compute locally for a time uniformly selected in the interval $(g - \frac{v}{2}, g + \frac{v}{2})$. By adjusting g we model parallel programs with different computational granularities and by varying v we change the degree of load-imbalance across processors. The communication phase consists of an opening barrier, followed by an optional sequence of pairwise communication events separated by small amounts of local computation, c, and finally an optional closing barrier. We consider two communication patterns: *Barrier* and *Transpose*. *Barrier* consists of only the opening barrier and thus contains no additional dependencies. This workload can can be used to analyze how our methodology responds to load imbalance. *Transpose* is a communication intensive workload. It tries to emulate the communication pattern generated by the FFT transpose algorithm [15], where each process accesses data on all other processes.

We consider three parallel jobs with the same computation granularity, load-imbalance and communication pattern arriving at the same time in the system. We fix the communication granularity, c, at 8 μsec. The number of communication/computation iterations is scaled so that each job runs for approximately 1 sec in a dedicated environment. The system consists of 32 processors and each job requires 32 processes (i.e. jobs are only time-shared).

4.2 The Simulation Model

The simulation tool that we used in the experimental evaluation is called SMART (Simulator of Massive ARchitectures and Topologies) [25], a flexible tool designed to model the fundamental characteristics of a massively parallel architecture.

The current version of SMART is based on the x86 instruction set. The architectural design of the processing nodes is inspired by the Pentium II family of processors [31]. In particular, it models a two level cache hierarchy with a write back L1 policy and non blocking caches. We assume a processor speed of 500 Mhz. In the experiments we will consider a network with 32 processors interconnected in a 5-dimensional cube topology with performance characteristics similar to those of Myrinet routing and network cards [3]. This network features a one-way data rate of about 1 Gbit/sec and a base network latency of few μsec. The simulator models at register level the congestion inside the network, at the network interface and the routing and flow control protocols.

The run time support running on this simulated platform includes a standard version of a significative subset of MPI-2 and a multitasking version of the same subset that performs the strobing algorithm at the end of each time-slice, as outlined in section 3. It is worth noting that the multitasking MPI-II version is much simpler than

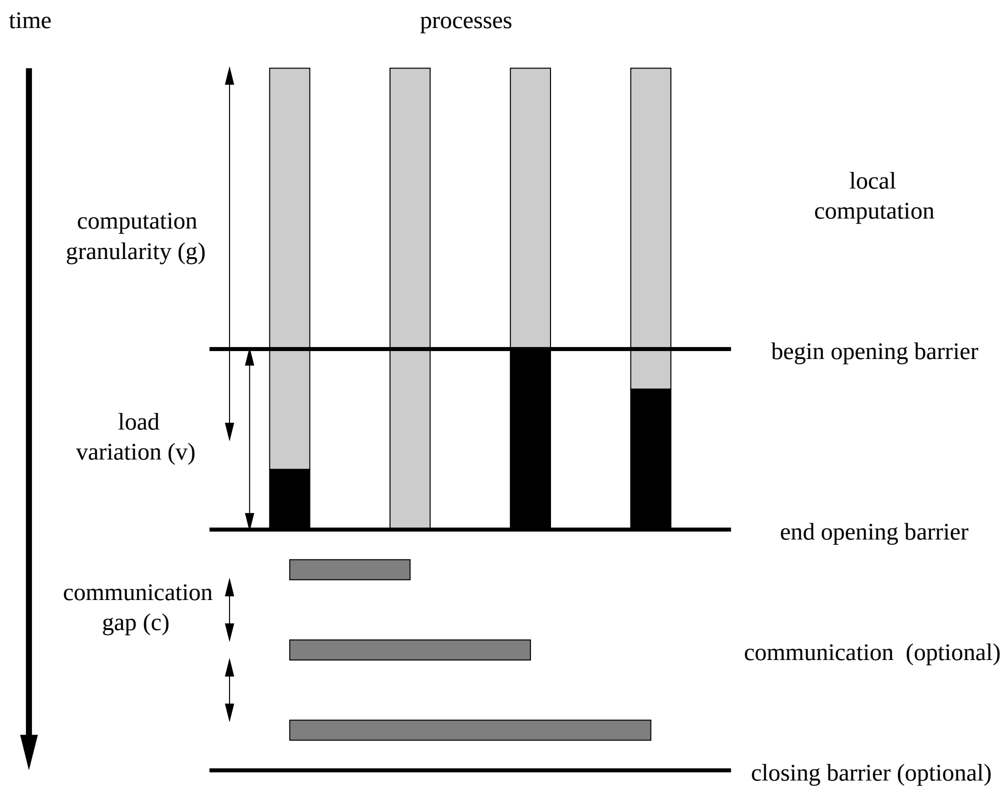

Figure 7: Each process of a parallel job executes on a separate processor and alternates between computation and communication phases. Processes compute for a mean time g before executing the opening barrier of the communication phase. The variation in computation across processes is uniformly distributed in $(0, v)$. Within the communication phase, each process computes for a small time c between events and the phase may close with a barrier.

the sequential one, because the buffering of the communication primitives greatly simplifies the run-time support.

4.3 Sensitivity Analysis

Figures 8 and 9 illustrate the communication and computation characteristics of our synthetic benchmarks as a function of the communication pattern, granularity, load-imbalance, time-slice duration and context switch penalty. Each bar shows the percentage of time spent in one of the following states, averaged over all processors: computing, context-switching and idling.

For each communication pattern, we analyze the Cartesian product of nine alternatives generated by considering time-slices of 500 μsec, 1 and 2 msec with a context switch penalty of 50, 100 and 200 μsec.

For each of these alternatives, we considers six groups of three bars. Each group has the same computation granularity, and the load imbalance is increased as a function of the granularity itself. We consider three cases: $v = 0$ (i.e. no variance), $v = g$ (in this case the variance is equal to the computational granularity) and $v = 2g$ (high degree of imbalance).

At the bottom of each figure we also report the breakdown for the same communication pattern when the workload is run in dedicated mode on the standard MPI-II run time support (i.e. a single job is run until completion without multitasking). A black square under a bar highlights the configurations where the multitasking approach gets better resource utilization than the standard approach.

By examining the breakdowns of each bar, we can see several important features. As the load imbalance of the program increases (i.e., moving to the right within each group of three bars with the same computational granularity) the idle time increases. For each group of 18 bars with the same time-slice and context switch penalty we reduce the computational grain size, going from left to right, from 50 msec to 100

μsec.

The time-slice length is critical parameter for the overall performance. A short time-slice can achieve a very good load balancing even in the presence of highly unbalanced jobs. The downside is that it amplifies the context switch latency. On the other hand, a long time-slice can virtually hide all the context switch latency, but cannot reduce the load imbalance, in particular when we have fine-grained computations.

In Figure 8 g) we can see that a relatively small time-slice coupled with a small context switch latency can get a high processor utilization, which is better than the one of the single job running in a dedicated environment (or, equivalently, the performance of zero-latency coscheduling) in eleven cases out of eighteen. Running a single job provides a slightly better (less than 10%) performance with perfect load balancing ($v = 0$) because we have to pay the context switch penalty without improving the load balance. On the other hand, in the presence of load imbalance, job multitasking can smooth the differences in load.

As a rule of thumb, multitasking gives good performance as long as the average computational grain size is larger than the time-slice and the time-slice, on its turn, is sufficiently larger than the context switch penalty.

Looking at Figure 8 and 9 we can also identify an important invariance. *When the average computational grain size is larger than the time-slice, the processor utilization is mainly influenced by the degree of imbalance.*

The experimental results show that the overall performance is sensitive to context switch latency. This implies that it is very important to minimize such latency. In these preliminary experimental results we did not take into account the effects of the memory hierarchy on the working sets of different jobs. As a consequence, the multitasking methodology requires a larger main memory, in order to avoid memory swapping. We consider this as the main limitation of our approach.

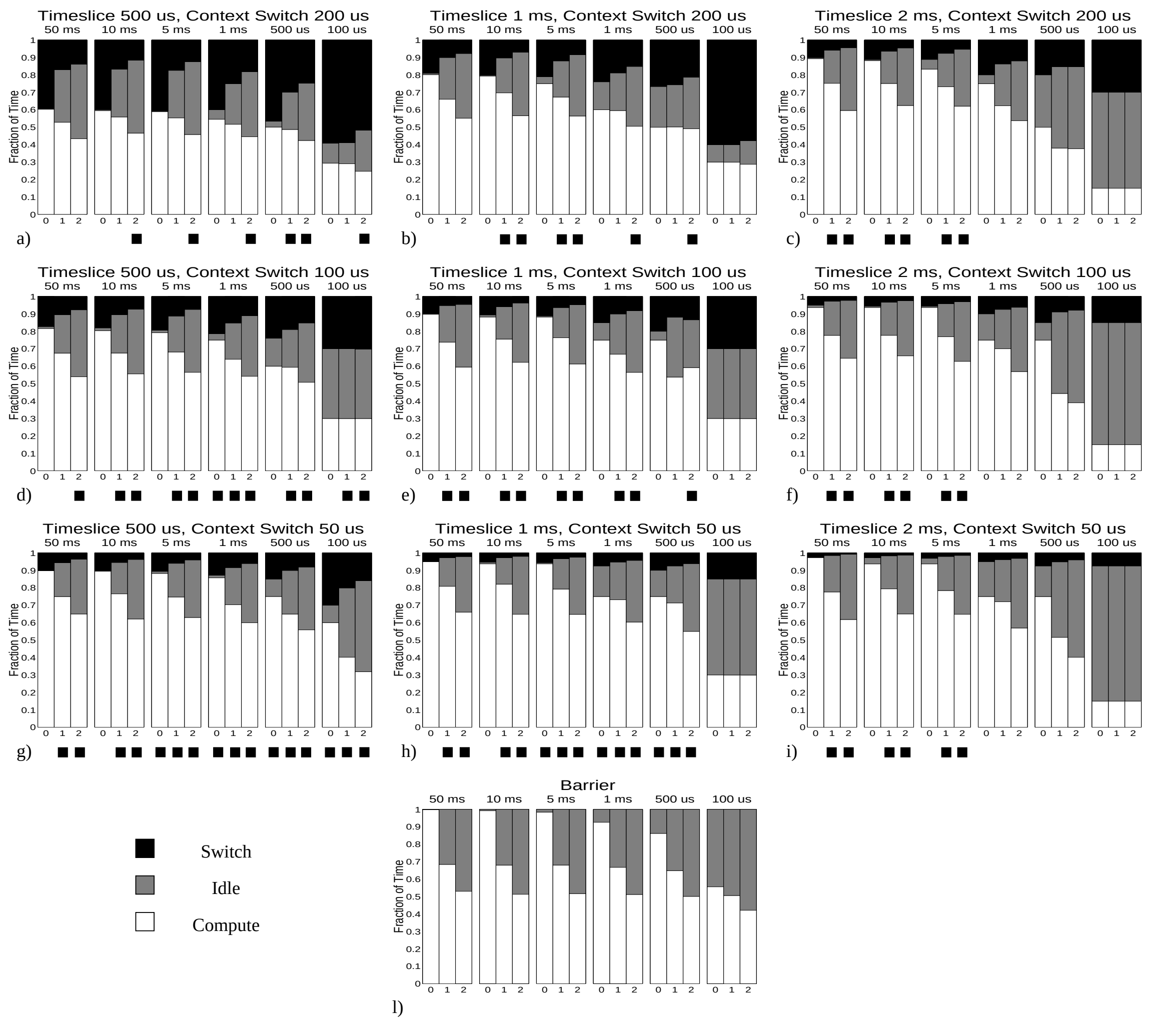

Figure 8: Execution characteristics as a function of computation granularity, load imbalance, time-slice length and context switch latency for the *Barrier* workload.

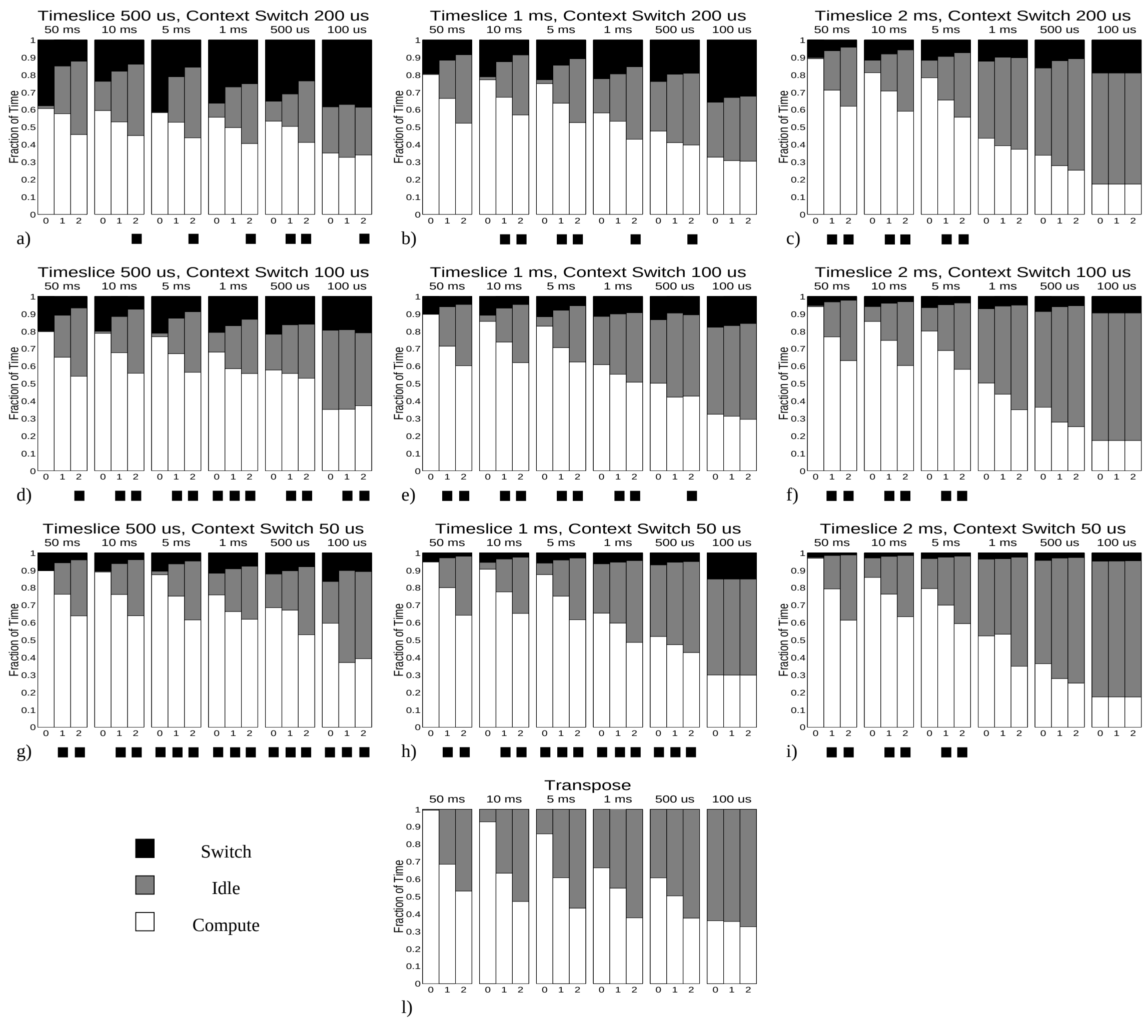

Figure 9: Execution characteristics as a function of computation granularity, load imbalance, time-slice length and context switch latency for the *Transpose* workload.

5 Conclusion and Future Work

In this paper we have presented a new methodology to multitask parallel jobs on a parallel computer. The methodology addresses the main limitation of explicit coscheduling, the high latency needed to perform a global context switch. Also, it provides a simple framework to increase the resource utilization, simplify the design of the run time support, increase the faults tolerance and perform effective global optimizations.

We tried to address the complexity of a huge design space using two families of syntethic workloads. The preliminary experimental results reported in this paper show that our methodology can provide a better resource utilization, in particular in the presence of load imbalance and communication intensive jobs.

We plan to extend these preliminary results by considering the effects of the memory hierarchy by considering real application rather than synthetic workloads and to implement in a Linux cluster a multitasking version of MPI-II.

References

[1] Andrea C. Arpaci-Dusseau, David Culler, and Alan M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of the 1998 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, Madison, WI, June 1998.

[2] Raul A. F. Bhoedjang, Tim Rühl, and Henri E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, November 1998.

[3] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawick, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, January 1995.

[4] Douglas C. Burger and David A. Wood. Accuracy vs. Performance in Parallel Simulation of Interconnection Networks. In *Proceedings of the 9th International Parallel Processing Symposium, IPPS'95*, Santa Barbara, CA, April 1995.

[5] Compaq, Intel, and Microsoft. The Virtual Interface Architecture (VIA) Specification. available at http://www.viarch.org.

[6] William J. Dally. Virtual Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

[7] William J. Dally and Charles L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[8] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of the 1996 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, Philadelphia, PA, May 1996.

[9] Dror G. Feitelson and Morris A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[10] Dror G. Feitelson and Larry Rudolph. Parallel job scheduling: issues and approaches. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[11] Dror G. Feitelson and Larry Rudolph. Toward Convergence in Job Schedulers for Parallel Supercomputers. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[12] Mike Galles. Spider: A High-Speed Network Interconnect. *IEEE Micro*, 17(1):34–39, January 1997.

[13] Alex Gerbessiotis and Fabrizio Petrini. Network Performance Assessment under the BSP Model. In *International Workshop on Constructive Methods for Parallel Programming, CMPP'98*, Marstrand, Sweden, June 1998.

[14] A. Gupta, A. Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference*, pages 120–132, May 1991.

[15] Anshul Gupta and Vipin Kumar. The Scalability of FFT on Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):922–932, August 1993.

[16] Adolfy Hoisie, Olaf Lubeck, and Harvey Wasserman. Scalability Analysis of Multidimensional Wavefront Algorithms on Large-Scale SMP Clusters. In *The Ninth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'99)*, Annapolis, MD, February 1999.

[17] Atsushi Hori, Hiroshi Tezuka, and Yukata Ishikawa. Overhead Analysis of Preemptive Gang Scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 217–230. Springer-Verlag, 1998.

[18] Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. Highly Efficient Gang Scheduling Implementation. In *Supercomputing 98*, Orlando, FL, November 1998.

[19] Morris A. Jette. Performance Characteristics of Gang Scheduling in Multiprogrammed Environments. In *Supercomputing 97*, San Jose, CA, November 1997.

[20] Vijay Karamcheti and Andrew A. Chien. Do Faster Routers Imply Faster Communication? In *First International Workshop, PCRCW'94*, volume 853 of *LNCS*, pages 1–15, Seattle, Washington, USA, May 1994.

[21] Walter Lee, Matthew Frank, Victor Lee, Kenneth Mackenzie, and Larry Rudolph. Implications of I/O for Gang Scheduled Workloads. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[22] Scott Mace. Commodity clusters. *Byte*, 23(4):87–90, April 1998.

[23] Shailabh Nagar, Ajit Banerjee, Anand Sivasubramaniam, and Chita R. Das. A Closer Look At Coscheduling Approaches for a Network of Workstations. In *Eleventh ACM Symposium on Parallel Algorithms and Architectures, SPAA'99*, Saint-Malo, France, June 1999.

[24] William E. Weihl Patrick Sobalvarro, Scott Pakin and Andrew A. Chien. Dynamic Coscheduling on Workstation Clusters. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 231–256. Springer-Verlag, 1998.

[25] F. Petrini and M. Vanneschi. SMART: A Simulator of Massive ARchitectures and Topologies. In *Proceedings of the International Conference on Parallel and Distributed Systems Euro-PDS'97*, June 1997.

[26] Fabrizio Petrini. Network Performance with Distributed Memory Scientific Applications. Submitted to the Journal of Parallel and Distributed Computing, September 1998.

[27] Fabrizio Petrini. Total-Exchange on Wormhole k-ary n-cubes with Adaptive Routing. In *Proceedings of the 12th International Parallel Processing Symposium, IPPS'98*, Orlando, FL, March 1998.

[28] Fabrizio Petrini and Marco Vanneschi. Efficient Personalized Communication on Wormhole Networks. In *The 1997 International Conference on Parallel Architectures and Compilation Techniques, PACT'97*, San Francisco, CA, November 1997.

[29] Fabrizio Petrini and Marco Vanneschi. SMART: a Simulator of Massive ARchitectures and Topologies. In *International Conference on Parallel and Distributed Systems Euro-PDS'97*, Barcelona, Spain, June 1997.

[30] Fabrizio Petrini and Marco Vanneschi. Latency and Bandwidth Requirements of Massively Parallel Programs: FFT as a Case Study. *Future Generation Computer Systems*, 1999. Accepted for publication.

[31] Tom Shanley. *Pentium Pro and Pentium II System Architecture*. Addison-Wesley, March 1998.

[32] D. B. Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Journal of Scientific Programming*, 1998.

[33] Patrick Sobalvarro and William E. Weihl. Demand-Based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors. In *Proceedings of the 9th International Parallel Processing Symposium, IPPS'95*, Santa Barbara, CA, April 1995.

[34] Kuniyasu Suzaki and David Walsh. Implementing the Combination of Time Sharing and Space Sharing on AP/Linux. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, 1998.